Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 6 (2011) 396–401

Procedia Computer Science

Complex Adaptive Systems, Volume 1
Cihan H. Dagli, Editor in Chief
Conference Organized by Missouri University of Science and Technology
2011- Chicago, IL

nesC-TinyOS model for parallel and distributed computation of max independent set by Hopfield network on wireless sensor network

Jiakai Li, Gursel Serpen

Electrical Engineering and Computer Science, University of Toledo, Toledo, Ohio 43606, USA

Abstract

This paper, the second one in a three-paper sequence, presents the nesC model of a Hopfield neural network configured for a static optimization problem, the maximum independent set, in fully parallel and distributed mode for TinyOS-based wireless sensor networks. Actual nesC code that implements the required neural computing functionality is presented. The graph representation of the maximum independent set problem is used as the basis for the topology of the Hopfield network as well as the wireless sensor network since each mote is conceived to house one neuron in order to facilitate fully parallel and distributed computation. The nesC implementation of a multitude of phases of computation is detailed including initialization of the neural network, relaxation, convergence detection, and solution detection all while the neural computations are performed on the wireless sensor network. Simulation of the presented nesC-TinyOS model is deferred to the third paper in the sequence.

Keywords: nesC, TinyOS, wireless sensor network, Hopfield neural network, static optimization, parallel and distributed computation, maximum independent set problem

1. Introduction

This paper, as the second one in sequence for a three-paper set, discusses the nesC-TinyOS model of fully parallel and distributed computation of solutions of maximum independent set problem through a Hopfield neural network embedded into a wireless sensor network as the hardware platform. The first paper [1] in the sequence presented the maximum independent set (MIS) problem definition, the Hopfield network computational model, the configuration of Hopfield network as a static optimizer to solve the MIS problem, and the mapping of Hopfield network to a wireless sensor network for fully parallel and distributed computation.

The MIS problem is defined as follows: assume a graph has a set of N vertices, V_i, i=1,2,…,N, and up to K edges, e_{ij}, $i,j=1,2,...,N$, where some of the edges may not exist. Consider a neural network with N neurons where outputs of neurons are represented by $z_1,\ldots,z_N$. Each neuron in the neural network will be mapped or correspond to a vertex in the graph. The MIS problem entails computing a maximum cardinality subset of a given graph such that there is no edge, e_{ij}=0, between any two vertices in the maximum cardinality subset. The motes in the wireless sensor network (WSN) correspond to vertices of the graph while one-hop communication links between motes correspond to edges between the corresponding vertices of the graph.

1877–0509 © 2011 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2011.08.074

Simulation of Hopfield neural network dynamics [2] on a digital system (i.e. a microcontroller on a sensor node) requires derivation of discrete-time equations using the continuous dynamics (given in Equation 2 in [1]). The specific form of discrete-time equation for the neuron dynamics will depend on the numerical integration method chosen. In general, the discrete time equation, in one of its simpler forms, will resemble the following construction:

$$u_i^{k+1} = h\left[u_i^k, \sum_{j=1}^{K} w_{ij} z_j^k + b_i \right] \text{ and } z_i^{k+1} = f(u_i^{k+1}) \text{ for } i = 1,2,....,K, \text{ and } k = 0,1,2,\ldots$$

where K is the number of neurons in the Hopfield network, z_i^k and u_i^k (while noting that certain neuron models are memoryless with respect to activation variable: the u_i^k term is missing as an argument to the h function) are the values of i-th neuron output and activation, respectively, at discrete time k (recursion index), w_{ij} is the weight between neurons z_i and z_j subject to $w_{ij} = w_{ji}$ and $w_{ii} = 0$, b_i is the external bias input for node z_i, $h(\cdot)$ is a function for implementing recursion, and $f(\cdot)$ is a nonlinearity - typically the sigmoid function with positive slope steepness value represented by λ. Equations of interest for the MIS problem mapped to Hopfield network dynamics were derived in [Serpen and Li, 2011] and are re-presented in Table 1 for reference.

Equation	***Description***	***Number***
$E = \frac{1}{2} g_a \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} e_{ij} z_i z_j + \frac{1}{2} g_b \sum_{i=1}^{N} (1 - z_i)$	Error function for maximum independent set (MIS) problem mapped to Hopfield network dynamics with N neurons	3
$w_{ij} = -2 g_a e_{ij}$ and $b_i = g_b$	Weight and external bias values definitions for a neuron	4
$2Mg_a > g_b$, where $g_a, g_b \in R^+$	Bounds on values of constraint weighting coefficients	5

Table 1. Error function, weight values, and bounds on constraint weighting parameters for MIS problem

2. Modeling in nesC-TinyOS Environment

Modeling the wireless sensor network and the embedded neural netwok algorithm was realized using the nesC (network embedded systems C) programming language on the TinyOS operating system [3]. TinyOS is open source with BSD-license. nesC is a component-based, event-driven programming language used to build applications for the TinyOS platform. nesC is built as an extension to the C programming language with components "wired" together to run applications on TinyOS. TinyOS version used was 1.1.11.3. TinyOS also requires the Java SDK: the default version of Java SDK (1.4.2) was utilized.

There are two kinds of motes in the wireless sensor network that serves as a parallel and distributed computing platform for the neural network, the supervisory mote and generic motes. Generic motes perform single neuron computations while the supervisory mote acts as a “supervisor” – it monitors, collects, processes and broadcasts the global information related to neural network computations. The supervisor mote may implement the neural computation functionality of a generic mote if necessary. The supervisory mote may have different hardware design with practically unlimited power and may be able to continuously operate for extended periods of time which would be unusual for typical generic motes. The supervisory mote is responsible broadcasting start and end times of different phases of neural computations including, but not limited to, initialization, convergence or neural network dynamics update, solution analysis and announcement, and re-initialization, where the latter is as needed.

2.1 Initialization - Neighborhood Discovery and Weight Vector Computation

Any two motes within one-hop neighborhood of each other (assuming communication in both directions exists), through a communication link points at existence of an edge between the two corresponding vertices associated with those two motes. Accordingly, each mote in the WSN (which maps to a vertex of the corresponding graph) must know its one-hop neighborhood or its connection topology. As presented in Equations 3 and 4 in Table 1, the connection topology information of each mote (which maps to a neuron in the neural network or vertex in the graph) is needed to be able to compute the weights or the weight vector for a given neuron. Each mote stores its one-hop neighborhood (adjacency or connectivity information) and weight vector for its neuron local memory.

The first step in the initialization is to find the one-hop neighbors of each mote. Therefore, it is necessary to implement the so-called “neighborhood discovery” phase. When the neighborhood discovery begins, every mote

sends out its own identity information (normally, its mote ID) through a so-called beacon packet in TinyOS environment. Motes are randomly selected for broadcasting based on the TinyOS B-MAC CSMA protocol [4]. Each mote must broadcast its beacon packet and broadcast only once during the neighborhood discovery period. After broadcasting its beacon packet, each mote also sends a report message to supervisory mote to indicate that it has broadcasted the beacon packet. Those who have already broadcasted may not send more beacon messages. Upon receipt of a beacon packet with a mote ID, a mote will store the one-hop neighbor's address in the incoming message in local memory and mark it as its neighbor. The supervisory mote maintains an array (vector) with a capacity that is equal to the number of generic motes in the sensor network to store the messages from the generic motes. As soon as all generic motes report that they already broadcasted beacon messages, the supervisory mote will broadcast a control message to the entire sensor network informing individual motes that the neighborhood discovery ended. Once the neighborhood discovery phase ends, each mote uses locally-stored one-hop neighbor list to construct its adjacency vector (adjacencyVector[] as a nesC static array),and compute the weight vector, (weightVector[] as a nesC static array) for its neuron through Equations 4 and 5 as in Table 1. The nesC code that implements neighborhood discovery and computation of weights through Equation 2 in [Serpen and Li, 2011] is presented in Figure 1. The nesC variable dGaWeightingCoefficient represents the constraint weighting coefficient g_a.

The next step in the initialization phase is to randomly set the neuron output values. This can be done locally also and a random value distributed uniformly in the closed interval [0.0,1.0] for unipolar output values can be generated for each neuron by the application program running on the associated mote. This value is stored in the local memory of the mote. The nesC implementation is shown in Figure 2.

```
// Perform neighbourhood discovery, determine one-hop neighbours, and compute weight vector for each mote
 if (initialized == FALSE) {
        Broadcast = TRUE;
         call SendMsg.send(TOS_BCAST_ADDR, sizeof(uint16_t), &beacon_packet);
          if (initReady == TRUE) {       //Initialize the weight vector through adjacency vector
                for (i = 0; i < NODE_COUNT; i++) {
                        weightVector[i] =  -2 * dGaWeightingCoefficient * adjacencyVector[i];
                         initialized = TRUE; }}}
```

Figure 1. TinyOS nesC code implementing neighborhood discovery and weight vector computation.

2.2 Timing of asynchronous update of neuron outputs

Each mote wakes up from sleep once every 1 time period (which is set to 1 second) through its dedicated timer. Timers in the wireless sensor network are not synchronized globally: every timer operates independent of others and yet has the same wakeup frequency of 1 time period. Accordingly, each neuron output gets updated once per 1 time period asynchronously of other neuron outputs. The timer code in nesC for each mote and the associated neuron is shown in Figure 3.

```
//initializes neuron outputs randomly to [0.0,1.0]
result_t InitForUpdate() {
    …
    //Randomly initialize neuron output through function
    //rand() that returns an integer in [0,65535]
    temp = call Random rand();
    //Distribute uniformly in the interval [0.0, 1.0]
    neuronOutput = temp/65535;
    …
    return SUCCESS;}
```

Figure 2. nesC code to initialize neuron outputs.

```
//implements timer and starts the application
command result_t StdControl.start() {
    //Initializes the variables for the first time
    InitForUpdate();
    //Starts the timer
    return call Timer.start(TIMER_REPEAT, 1000);}

//Stops the  timer and terminates the application
command result_t StdControl.stop() {
    return call Timer.stop();  }
```

Figure 3. nesC code for timer implementation.

After the initialization phase is complete, every neuron has its own initial randomly-generated output value, weight vector values, and the bias value defined, where the latter is computed through Equation 4 in Table 1, in the local memory of the associated mote. Upon entry into the convergence phase, motes wake up per the schedule of their timers and update the output of their neurons and broadcast the newly-updated neuron output value to their one-hop neighbors. One-hop neighbors store the received neuron output value in a local nesC array entitled

neighborsOutput[] as shown in Figure 4. Those same neighbors will consequently start updating the output value of their own neurons using the newly-updated values in the neighborsOutput[] array.

2.3 Neuron output update and convergence

Each mote computes the input for its neuron as shown in Figure 4. The neuron output is computed using locally stored network input as shown in Figure 5. Each generic mote computes the neuron output and calculates the difference between this newly computed output value and the most recent value which is stored locally. The difference value will be one of -1, 0, or 1 for a discrete Hopfield network: a 0 value indicates that neuron output stayed the same while either -1 or 1 will indicate that the neuron output changed. The convergence detection algorithm implemented on the supervisory mote needs to know if all neuron outputs stopped changing and hence collects this change information from all generic motes that participated in the neural network computations. Accordingly, each mote needs to compare the two most recent output values for its neuron: the outcome will be either a 0 (which signifies no change) or a 1 (which signifies a change). Once the mote calculates the neuron output difference value, it sends it to the supervisory mote if convergence has not been achieved yet where the latter would be signaled to all generic motes by the supervisory mote. There are a number of packet types being exchanged among motes and are numbered as listed in Table 2. The nesC implementation of this computation process for the neuron output is illustrated in Figure 5.

Packet Type	***Interpretation***
1	A generic mote is broadcasting its updated neuron output value to its one-hop neighbors.
2	A generic mote is sending the difference information for its neuron output values to the supervisory mote.
3	The supervisory mote is broadcasting “Hopfield neural network dynamics convergence status” message.
4	A generic mote is sending the locally-computed energy-related information to the supervisor mote.
5	The supervisory mote is broadcasting “MIS solution status” message to the entire sensor network.
6	A generic mote is announcing itself as a cluster head.
7	A generic mote is sending a report message to indicate SM that it has broadcasted.
8	The supervisory mote is broadcasting “Neighborhood discovery ends” message to entire sensor network.

Table 2. TinyOS-nesC Packet Types and Interpretation for Hopfield-MIS Application

```
result_t ComputeNeuronInput() {
    int i; NeuronInput = 0.0;
    //if convergence is not achieved yet, update.
    if (!convergenceAchieved) {
    // Update the neuron network input
        for ( i = 0; i < NODE_COUNT; i++ )
          NeuronInput += weightVector[i] *
          neighborsOutput[i];
          //Store the result in NetValue variable
          NetValue = NeuronInput;
          //Add the excitation bias value to it.
          NeuronInput += dExcitationBias; }
     return SUCCESS; }
```

Figure 4. nesC implementation of neuron input computation

```
result_t ComputeNeuronOutput(float Input) {
    //Before updating, save the last neuron output value.
    NeuronOldOutput = NeuronOutput;
    if (Input > 0.0) NeuronOutput = 1;
    else NeuronOutput = 0;
    if (NeuronOldOutput > 1) diff = 1;
    else //store the difference of two updates
          diff = NeuronOutput - NeuronOldOutput;
    //diff is uint, and note that diff = 0 - 1 = 65535
    //Now the correction to this is to set diff to 1.
    if (diff == 65535)  diff = 1;
        pending = FALSE; //Now output update is complete.
    //If no convergence yet, send the diff value to SM.
    if (!achievedConvergence) {
      PacketType = 2; call IntOutput.output(diff); }
    return SUCCESS; }
```

Figure 5. nesC implementation of neuron output computation

2.4 Convergence

Hopfield neural network dynamics converge to a fixed point when all neuron outputs stop changing by an appreciable amount. One method for detecting convergence is to collect output change information from all neurons through the supervisory mote (SM). If the sum of output change value for all neurons is less than a preset small positive threshold value (which is 0 for discrete neuron dynamics), then convergence can be assumed.

The supervisory mote acts as a global manager and is responsible for global information management, processing and interpretation. It collects the difference information sent from all generic motes participating in the Hopfield neural network computation and stores them in its local memory. Each time the supervisory mote wakes up it performs convergence check by simply summing over all difference values sent by participating motes. This is implemented through the nesC code presented in Figure 6.

Each generic mote sends the two-update-cycle difference information to the supervisory mote after updating the output of its neuron. This difference value is either 0 or 1. The supervisory mote after each wakeup adds up values of elements of this vector. If the summation is more than 0, at least one of the neuron outputs has changed: the convergence has not been achieved yet. If the summation is 0, that is to say there is no difference between two consecutive update cycles for all the neuron outputs in the network, then the Hopfield neural network must have converged to a fixed point. Consequently, the supervisory mote broadcasts a message to every generic mote that participates in the neural computation, and tells them that the Hopfield network has converged and everyone should stop updating. This is implemented through the nesC code presented in Figure 7. The motes who received the convergence notice next compute local energy values and send them to the supervisory mote that will then sum them to determine if the fixed point is a solution of the MIS problem. The formula used for these calculations (partial energy or error function value locally computed by each mote for its corresponding neuron) is presented in Equation 6:

$$E_i = z_i \sum_{i=1}^{N} w_{ij} z_j, \text{ for } i = 1,2,\ldots,N. \tag{6}$$

```
event TOS_MsgPtr ReceiveMsg receive (TOS_MsgPtr
        recv_packet) {
        …
        HNMsg1 *RecvMessage =  (HNMsg1 *)
                                        recv_packet->data;
        //If packet type= 2, current packet is the difference.
        if (RecvMessage->type == 2){
              differenceVector[RecvMessage->src] =
              RecvMessage->val;
              convergenceAchieved = Convergence(); }
        …
        return SUCCESS; }
```

Figure 6. nesC code executing on supervisory mote to record incoming packets for output difference

```
result_t Convergence() {
   int i, sum; sum = 0;
   if (SNmode == TRUE) {  // executed by SM only!
   for (i = 0; i < NODE_COUNT; i++)
        sum = sum + diffVector[i];
   if (sum == 0) {  //All neuron outputs stopped changing.
          achievedConvergence = TRUE;
          //Network converged and ready for solution check.
          PacketType = 3;
          //An arg value of 2 indicates convergence.
          call IntOutput.output(2);
          return SUCCESS;}
   else  return FAIL; } return FAIL; }
```

Figure 7. nesC code implementation of convergence detection on supervisory mote (SM)

2.5 Solution Identification

A successful convergence to a fixed point by Hopfield network dynamics does not necessarily mean a solution has been computed. The supervisory mote collects the locally computed energy-related information and sums them to compute the error function for the maximum independent set (MIS) problem as in Equation 3 in Table 1. In the case of MIS it is desirable to activate as many neurons as possible without violating the constraint that no two neurons that are connected or one-hop neighbors (equivalently two corresponding vertices in the graph have an edge between them) can be active. The first term in MIS error function given in Equation 3 must have a value of zero for a solution while the second term might be any (positive) value. Since the quality of solution is determined by the search algorithm being used, which is gradient descent for the case of Hopfield network dynamics, if and when a solution is found, there is no guarantee that it is optimum or maximum. Therefore, all that can be done is to check if the fixed point is a solution. This can be achieved by making sure that the quadratic term in the MIS error function has a value of zero. The calculation performed by the supervisory mote for determining if a fixed point is a solution or not based on locally-computed energy-related information. The supervisory mote needs two values from each generic mote participating in the neural network computations: these are the most recent values of output and network input immediately prior to receiving the convergence achieved message from the supervisory mote. Upon receipt from all generic motes, the supervisory mote employs Equation 7 to compute the overall network-wide error function which is implemented in nesC code as in Figure 8.

$$\frac{1}{2}\sum_{i=1}^{N}\sum_{\substack{j=1\\ j\neq i}}^{N} w_{ij} z_i z_j = \frac{1}{2}\left[z_1 \sum_{j=1}^{N} w_{ij} z_j + z_2 \sum_{j=1}^{N} w_{ij} z_j + \cdots + z_N \sum_{j=1}^{N} w_{ij} z_N \right] = z_1 \times net_1 + z_2 \times net_2 + \cdots + z_N \times net_N \quad (7)$$

If the summation due to quadratic error term is not equal to 0, the currently converged fixed point is not a solution. The supervisory mote sends a broadcast message to all the motes (with a value of 0) to signify that the fixed point is not a solution. The Hopfield neural network needs to be reset and a new search through a new convergence cycle will need to be performed starting with new randomly initialized neuron outputs. If the summation is 0, that is to say a solution is found, the supervisory mote sends a broadcast message to all the other motes with the value of 1 which indicates that a solution has been found. The nesC code that implements solution identification by the supervisory mote is presented in Figure 9.

```
event TOS_MsgPtr ReceiveMsg receive(TOS_MsgPtr
                                    recv_packet) {
  HNMsg1 *RecvMessage = (HNMsg1 *)recv_packet->data;
  //If packet type=3, current packet is convergence info.
  if (RecvMessage->PacketDataType == 3){
     convergenceFound = TRUE;
     for (i = 0; i < NODE_COUNT; i++)
     netValue += fWeightVector[i] * fNeighborsOutput[i];
     localEnergy = neuronOutput * netValue;
     if (localEnergy < 0) {
        bNeuronEnergySign = FALSE;
        neuronEnergy = localEnergy * -1000; }
     else {
        bNeuronEnergySign = TRUE;
        neuronEnergy = localEnergy * 1000; }
     packetType = 4;
     all IntOutput.output(neuronEnergy); }
  return SUCCESS; }
```

Figure 8. nesC code demonstrating calculation of partial error through Equation 4 on a generic mote

```
result_t SolutionFound(){
  int i,EnergySum; EnergySum = 0;
  if (SNmode == TRUE) {
     if (GetConvergence == SUCCESS &&
                                   EnergySum == 0){
        //Solution is found!
        GetSolution = TRUE;
        PacketType = 5;
        call IntOutput.output(1);
        return SUCCESS; }
     else {//solution is not found!  Reset the neural network.
        GetSolution = FALSE;
        PacketType = 5;
        call IntOutput.output(0);
        reset = TRUE;
        initForUpdate();
        return FAIL; } }
  return FAIL; }
```

Figure 9. nesC code implementation of solution identification on supervisory mote

3. Conclusions

This paper presented nesC-TinyOS realization of a Hopfield neural network configured as a static optimizer for the maximum independent set problem and embedded into a wireless sensor network as a hardware computing platform for fully parallel and distributed computation. This is the second paper in a three-paper sequence that covers mapping a Hopfield neural network configured for static optimization to wireless sensor network for parallel and distributed computation (the first paper), modeling the neural wireless sensor network on nesC-TinyOS platform, and simulation using the TOSSIM (reported in the third paper in the sequence [5]).

References

[1] Serpen, G. and Li, J., "Parallel and distributed computation of maximum independent set by Hopfield neural net embedded into a wireless sensor network", Proc. of Complex and Adaptive Systems Conference, Chicago, 2011.

[2] Hopfield J. J., and Tank, D. W. "Neural Computation of Decision in Optimization Problems," *Biological Cybernetics,* Vol. 52, pp. 141-152, 1985.

[3] Gay,D., Levis, P. Behren, R., Welsh, M., Brewer, E. and Culler.D "The nesC language: A holistic approach to networked embedded systems" *in Proc. of the ACM Conf. on Programming Language Design and Implementation* NY, USA 2003

[4] Polastre, J., Hill, J. and Culler, D. "Versatile Low Power Media Access for Wireless Sensor Networks" *in Proceeding of SenSys '04 Proc. of the 2nd international conference on Embedded networked sensor systems,* New York, NY, USA 2004

[5] Li, J. and Serpen, G., "TOSSIM simulation of wireless sensor network serving as hardware platform for Hopfield neural net configured for max independent set," Proc. of Complex and Adaptive Systems Conference, Chicago, 2011.